地图缩小：大型集群上的简化数据处理

Jeffrey Dean和Sanjay Ghemawat

jeff@google.com，sanjay@google.com

谷歌公司。

摘要

Map Reduce是用于处理和生成大型数据集的编程模型和相关实现。 用户指定一个映射函数，该函数处理一个键/值对以生成一组中间键/值对，以及一个减少函数，该函数合并与同一中间键相关的所有中间值。 在这个模型中，许多现实世界的任务是可以解释的，如本文所示。

以这种功能样式编写的程序会自动并行化并在大型商品机器集群上执行。 运行时系统负责划分输入数据的细节，在一组机器上调度程序的执行，处理机器故障，以及管理所需的机器间通信。 这使得没有任何并行和分布式系统经验的程序员可以轻松地利用大型分布式系统的资源。

我们的Map Reduce实现运行在一个大型商品机器集群上，并且具有很高的可伸缩性：一个典型的Map Reduce计算在数千台机器上处理许多TB的数据。 程序员发现这个系统很容易使用：数百个Map Reduce程序已经实现，每天在Google的集群上执行超过1000个Map Reduce作业。

1 导言

在过去的五年中，作者和谷歌的许多其他人已经实现了数百种特殊用途的计算，处理大量的原始数据，如爬行文档、Web请求日志等，以计算各种派生数据，如倒排索引、Web文档图形结构的各种表示、每个主机爬取的页数摘要、一组最常见的查询给定的一天等。 大多数这样的计算在概念上是简单的。 然而，输入数据通常很大，计算必须分布在数百或数千台机器上，才能在合理的时间内完成。 如何并行计算、分发数据和处理故障等问题，共同掩盖了用大量复杂代码处理这些问题的原始简单计算。

作为对这种复杂性的反应，我们设计了一个新的抽象，它允许我们表达我们试图执行的简单计算，但隐藏了库中并行化、容错、数据分发和负载平衡的混乱细节。 我们的抽象灵感来自地图，减少了Lisp和许多其他功能语言中的原语。 我们意识到，我们的大多数计算都涉及对输入中的每个逻辑“记录”应用映射操作，以便计算一组中间键/值对，然后对共享相同键的所有值应用减少操作，以便适当地组合派生数据。 我们使用具有用户指定映射和减少操作的功能模型，允许我们轻松地并行大型计算，并使用重新执行作为容错的主要机制。

这项工作的主要贡献是一个简单而强大的接口，它能够自动并行化和分发大规模计算，并结合该接口的实现，在大型商品PC集群上实现高性能。

第二节描述了基本的编程模型，并给出了几个例子。 第三节描述了针对基于集群的计算环境定制的Map Reduce接口的实现。 第四节描述了我们发现有用的编程模型的几个改进。 第五节对我们实现各种任务的性能进行了测量。 第六节探讨了Map Reduce在谷歌内部的使用，包括我们使用它作为基础的经验

重写我们的生产索引系统。 第七节讨论了相关工作和今后的工作。

2 编程模型

计算需要一组输入键/值对，并生成一组输出键/值对。 Map Reduce库的用户将计算表示为两个函数：Map和Reduce。

由用户编写的映射获取输入对并生成一组中间键/值对。 Map Reduce库将与同一中间键I关联的所有中间值组合在一起，并将它们传递给Reduce函数。

还由用户编写的Reduce函数接受中间密钥I和该密钥的一组值。 它将这些值合并在一起，形成一组可能较小的值。 通常每个Reduce调用只产生零或一个输出值。 中间值通过迭代器提供给用户的Reduce函数。 这允许我们处理太大而不适合内存的值列表。

2.1 例如

考虑计算大量文档集合中每个单词出现次数的问题。 用户将编写类似于以下伪代码的代码：

```
映射(String键，String值)：
  //密钥：文档名称
  //值：每个字w的文档内容值：
    中间(w，“1”)；

减少（字符串键，迭代器值）：
  //重点：一个词
  //值：计数列表int结果=0；
  对于值中的每个v：结果+=解析
    Int(V)；
  省略（作为字符串(结果)）；
```

映射函数发出每个单词加上相关的出现计数（在这个简单的例子中只有‘1）。 减少函数将为特定单词发出的所有计数相加在一起。

此外，用户编写代码以填写带有输入和输出文件名称的mapreduce规范对象，以及可选的调优参数。然后用户调用Map Reduce函数，将其传递给规范对象。 用户的代码与Map Reduce库(在C++中实现)链接在一起)。 附录A包含此示例的完整程序文本。

2.2 类型

尽管以前的伪代码是用字符串输入和输出编写的，但从概念上讲，用户提供的映射和减少函数具有关联类型：

地图	(k1，v1)	→清单(k2，v2)
减少	(k2，清单(v2))	→列表(V2)

I.，输入键和值是从与输出键和值不同的域中提取的。 此外，中间键和值来自与输出键和值相同的域。

我们的C++实现将字符串传递给用户定义的函数，并将其留给用户代码，以便在字符串和适当的类型之间进行转换。

2.3 更多的例子

下面是一些有趣的程序的简单例子，它们可以很容易地表示为Map Reduce计算。

分布式Grep：映射函数如果与提供的模式匹配，则会发出一条线。 减少函数是一个标识函数，它只将提供的中间数据复制到输出。

URL访问频率计数：地图功能处理网页请求日志并输出URL，1。 减少函数将同一URL的所有值相加，并发出一个URL，总计数对。

反向Web链接图：映射函数输出目标、每个链接的源对对到名为源的页面中找到的目标URL。 减少函数连接与给定目标URL关联的所有源URL的列表，并发出对：(target，list(source))

每个主机的术语向量：术语向量将文档或一组文档中发生的最重要的单词总结为单词、频率对的列表。 映射函数为每个输入文档发出一个hostname，术语向量对(其中hostname是从文档的URL中提取出来的)。 减少函数传递给给定主机的所有每个文档项向量。 它将这些项向量加在一起，扔掉不常见的项，然后发出最后的项
（主机名，术语向量）对。

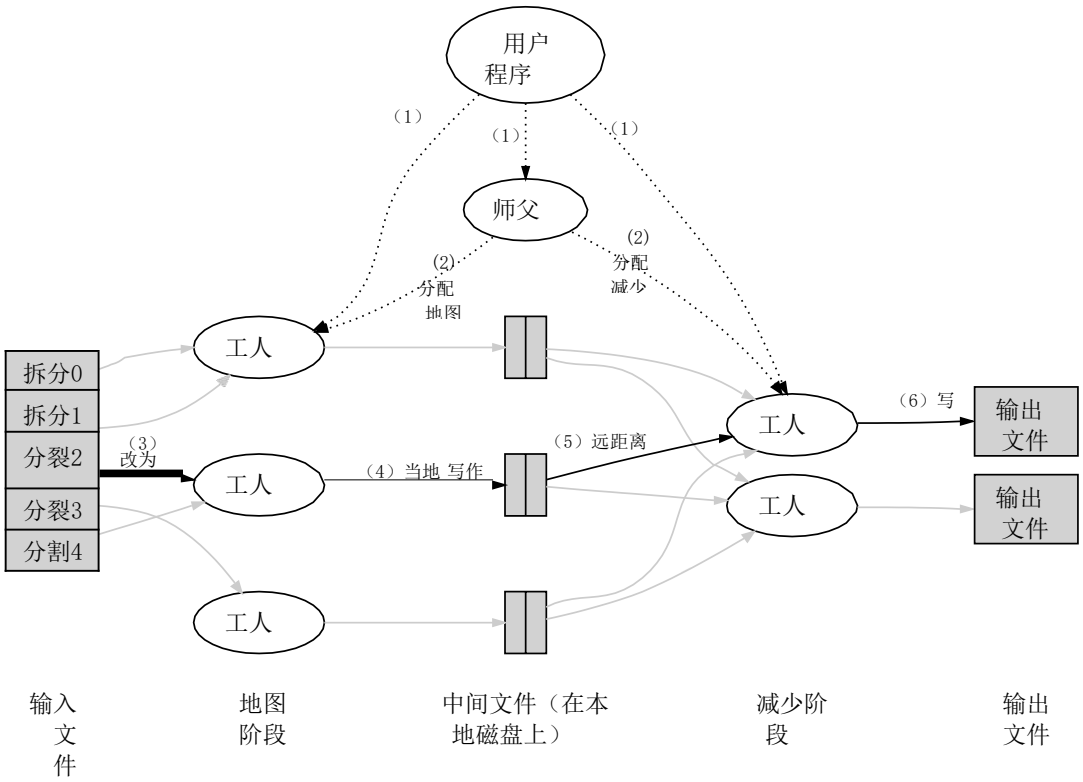


图1：执行概述

倒置索引：映射函数解析每个文档，并发出一系列Word、文档ID对。 减少函数接受给定字的所有对，对相应的文档ID进行排序，并发出单词、列表(文档ID)对。 所有输出的集合对形成一个简单的倒置索引。 它很容易增强此计算以跟踪单词位置。

分布式排序：映射函数从每个记录中提取密钥，并发出密钥，记录对。 还原函数发射所有对不变。这个公司-
这取决于4.1节中描述的分区设施和4.2节中描述的排序属性。

3 执行情况

Map Reduce接口的许多不同实现是可能的。 正确的选择取决于环境。 例如，一个实现可能适合于小型共享内存机器，另一个用于大型NUMA多处理器，另一个用于更大的网络机器集合。

本节描述了针对谷歌广泛使用的计算环境的实现：

大型商品PC集群与交换以太网[4]连接在一起。 在我们的环境中：

(1) 机器通常是运行Linux的双处理器x86处理器，每台机器有2-4GB的内存。

(2) 商品网络硬件被使用-通常是100兆位/秒或1千兆位/秒在机器级别，但平均在整个二分带宽上要少得多。

(3) 集群由数百或数千台机器组成，因此机器故障是常见的。

(4) 存储由直接连接到单个机器的廉价IDE磁盘提供。 内部开发[8]分布式文件系统用于管理存储在这些磁盘上的数据。文件系统使用复制在不可靠的硬件之上提供可用性和可靠性。

(5) 用户向调度系统提交作业。 每个作业由一组任务组成，并由调度程序映射到集群内的一组可用机器。

3.1 执行概述

通过自动分区输入数据，Map调用分布在多台机器上

变成一组M分裂。 输入分裂可以由不同的机器并行进行。 通过使用分区函数将中间密钥空间划分为R块来分配减少调用(例如。，
散列（键）modR)。 分区(R)和
分区函数由用户指定。 图1显示了Map Reduce OP-
的总体流程-
在我们的实施中。 当的用户程序
调用Map Reduce函数，会发生以下操作序列（图1中的编号标签对应于下面列表中的数字）：

1. 用户程序中的Map Reduce库首先将输入文件分割成M块，通常为每块16兆字节到64兆字节(MB)（由用户通过可选参数控制）。 然后它在一组机器上启动程序的许多副本。

2. 程序的一个副本是特殊的-主人。 其余是由师傅分配工作的工人。 有M图任务和R减小任务要分配。 主选择空闲工作人员，并为每个人分配一个映射任务或减少任务。

3. 分配地图任务的工作人员读取相应输入拆分的内容。 它解析输入数据中的键/值对，并将每对传递给用户定义的Map函数。 映射函数产生的中间键/值对在内存中缓冲。

4. 定期地，缓冲对被写入本地磁盘，通过分区函数划分为R区域。 这些缓冲对在本地磁盘上的位置被传递回主，主方负责将这些位置转发给减少工作人员。

5. 当主程序通知减少工作人员有关这些位置时，它使用远程过程调用从地图工作人员的本地磁盘读取缓冲数据。 当一个reduce工作人员读取所有中间数据时，它将其按中间键排序，以便将同一键的所有出现都分组在一起。 需要排序，因为通常有许多不同的键映射到相同的减少任务。 如果中间数据量太大，无法适应内存，则使用外部排序。

6. 减少工作人员迭代排序的中间数据，对于遇到的每个唯一中间键，它将键和相应的中间值集合传递给用户的减少函数。 减少函数的输出被附加到此减少分区的最终输出文件中。

7. 当所有地图任务和减少任务都已完成时，主程序将唤醒用户程序。 此时，用户程序中的Map Reduce调用返回到用户代码。

成功完成后，mapreduce执行的输出在R输出文件中可用（每个减少任务一个，文件名由用户指定）。通常，用户不需要将这些R输出文件组合成一个文件-它们通常将这些文件作为输入传递给另一个Map Reduce调用，或者从能够处理被划分为多个文件的输入的另一个分布式应用程序中使用它们。

3.2 掌握数据结构

主保持多个数据结构。 对于每个映射任务和减少任务，它存储状态（空闲、正在进行或完成)和工作机器的标识(对于非空闲任务）。

主机是从映射任务传播中间文件区域位置以减少任务的管道。 因此，对于每个完成的地图任务，主存储图任务产生的R中间文件区域的位置和大小。随着地图任务的完成，将收到对该位置和大小信息的更新。 这些信息被逐步推送给那些正在执行减少任务的工人。

3.3 容错

由于Map Reduce库是为了帮助使用数百或数千台机器处理大量数据而设计的，因此库必须优雅地容忍机器故障。

工人失败

师傅定期给每个工人打针。 如果在一定的时间内没有收到工人的响应，主将工人标记为失败。 工作人员完成的任何映射任务都被重置回初始空闲状态，因此有资格对其他工作人员进行调度。 类似地，在失败的工作人员上进行的任何映射任务或减少任务也被重置为空闲，并有资格重新安排。

完成的映射任务在故障时重新执行，因为它们的输出存储在故障机器的本地磁盘上，因此无法访问。 完成的减少任务不需要重新执行，因为它们的输出存储在全局文件系统中。

当地图任务首先由工人A执行，然后由工人B执行(因为A失败)时，所有

执行减少任务的工人将被通知重新执行。 任何尚未从工人A读取数据的减少任务都将从工人B读取数据。

Map Reduce能够抵御大规模的工人故障。 例如，在一次Map Reduce操作中，运行集群上的网络维护导致一次有80组机器无法访问几分钟。 Map Reduce主程序简单地重新执行无法到达的工作机器所完成的工作，并继续向前推进，最终完成Map Reduce操作。

失败大师

很容易使上面描述的主数据结构的主写周期检查点。 如果主任务死亡，则可以从最后一个检查指定状态启动新副本。 然而，鉴于只有一个主程序，它的失败是不可能的；因此，如果主程序失败，我们当前的实现将中止Map Reduce计算。 客户端可以检查此条件，并在需要时重试Map Reduce操作。

错误存在中的语义

当用户提供的映射和减少运算符是其输入值的确定性函数时，我们的分布式实现产生的输出与整个程序的非故障顺序执行所产生的输出相同。

我们依靠映射的原子提交和减少任务输出来实现这个属性。 每个正在进行的任务将其输出写入私有临时文件。 减少任务产生一个这样的文件，地图任务产生R这样的文件（每个减少任务一个）。 当地图任务完成后，工作人员向主机发送消息，并在消息中包含R临时文件的名称。 如果主服务器接收已完成的映射任务的完成消息，则忽略该消息。 否则，它将在主数据结构中记录R文件的名称。

当减少任务完成后，减少工作人员将其临时输出文件原子地重命名为最终输出文件。 如果在多台机器上执行相同的减少任务，则将对相同的最终输出文件执行多个重命名调用。 我们依赖于底层文件系统提供的原子重命名操作，以保证最终的文件系统状态只包含减少任务的一次执行所产生的数据。

我们的绝大多数映射和减少运算符都是确定性的，在这种情况下，我们的语义等价于顺序执行，这使得它非常

程序员很容易对他们的程序行为进行推理。 当映射和/或减少运算符是非确定性的时，我们提供了较弱但仍然合理的语义。 在存在非确定性运算符的情况下，特定减少任务R的输出$_1$ 是相当的去用于R的输出$_1$ 由顺序执行产生的非确定性程序。 但是，不同减少任务R的输出$_2$ 可能对应于产出为R_2 由非确定性的不同顺序执行产生 程序。

考虑映射任务M，减少任务R_1 和R_2。 让e(R_i)是R的执行$_i$ 承诺的（正是这样的执行）。 较弱的语义产生是因为e(R_1可能已经读取了M和e(R)的一次执行产生的输出$_2$)可能已经读取了输出 生产 通过 a 不同的 执行 的 *m*.

3.4 地点

网络带宽是我们计算环境中相对稀缺的资源。 我们利用输入数据(由GFS[8]管理)存储在构成集群的机器的本地磁盘上这一事实来节省网络带宽。 GFS将每个文件划分为64MB块，并在不同的机器上存储每个块的几个副本(通常为3个副本。 Map Reduce master考虑到输入文件的位置信息，并尝试在包含相应输入数据副本的机器上调度映射任务。 否则，它试图在该任务输入数据的副本附近（例如，在与包含数据的机器相同的网络交换机上的工人机器上）调度映射任务）。 当对集群中相当一部分工作人员运行大型Map Reduce操作时，大多数输入数据都是本地读取的，并且不消耗网络带宽。

3.5 任务粒度

我们将地图相位细分为M片，将还原相位细分为R片，如上所述。 理想情况下，M和R应该比工人机器的数量大得多。 让每个工作人员执行许多不同的任务可以改善动态负载平衡，并在工作人员失败时加快恢复速度：它已经完成的许多映射任务可以分散到所有其他工作人员机器上。

在我们的实现中，M和R可以有多大是有实际意义的，因为主程序必须做出O(M+R)调度决策，并保持O(M R)状态在内存中，如上所述。 但是，内存使用的常量因素很小：状态的O(MR)部分由每个映射任务/减少任务对大约一个字节的数据组成。 ） *

此外，R经常受到用户的限制，因为每个减少任务的输出最终都在一个单独的输出文件中。 在实践中，我们倾向于选择M，使每个单独的任务大约是16MB到64MB的输入数据（这样上面描述的局部性优化是最有效的），我们使R成为我们期望使用的工人机器数量的一小部分。 我们经常用M=200，000和R（5000）执行Map Reduce计算，使用2000台工人机器。

3.6 备份任务

延长Map Reduce操作总时间的一个常见原因是“分层器”：一台机器需要异常长的时间来完成最后几个映射中的一个或减少计算中的任务。 漂流者的出现有很多原因。 例如，磁盘不良的机器可能会经常出现可纠正的错误，从而将其读取性能从30MB/s降低到1MB/s。 集群调度系统可能已经在机器上调度了其他任务，由于CPU、内存、本地磁盘或网络带宽的竞争，导致它执行Map Reduce代码的速度更慢。我们最近遇到的一个问题是机器初始化代码中的一个错误，它导致处理器缓存被禁用：受影响的机器上的计算速度减慢了100多倍。

我们有一个普遍的机制来缓解散兵问题。 当Map Reduce操作接近完成时，主程序会安排剩余正在执行任务的备份执行。 每当主执行或备份执行完成时，任务被标记为已完成。 我们已经调整了这个机制，使它通常将操作使用的计算资源增加不超过百分之几。 我们发现，这大大减少了完成大型Map Reduce操作的时间。 例如，5.3节中描述的排序程序在禁用备份任务机制时需要花费44%的时间才能完成。

4 装修

虽然简单地编写Map和Reduce函数提供的基本功能足以满足大多数需求，但我们发现一些扩展是有用的。 这些将在本节中描述。

4.1 分区函数

Map Reduce的用户指定他们想要的Reduce任务/输出文件的数量(R)。 数据通过这些任务使用分区函数进行分区中间键。 提供使用散列的默认分区函数(例如。“散列（键）mod R”)。 这往往导致相当平衡的分区。 然而，在某些情况下，用密钥的其他函数来划分数据是有用的。 例如，有时输出键是URL，我们希望单个主机的所有条目最终都在同一个输出文件中。 为了支持这种情况，Map Reduce库的用户可以提供一个特殊的分区功能。 例如，使用“散列(Hostname(urlkey)modR”作为分区函数会导致来自同一主机的所有URL最终出现在同一输出文件中。

4.2 订购担保

我们保证在给定的分区内，中间键/值对按增加键的顺序处理。 这种排序保证使每个分区很容易生成排序输出文件，当输出文件格式需要支持按键进行高效的随机访问查找时，这很有用，或者输出的用户发现对数据进行排序很方便。

4.3 组合功能

在某些情况下，每个映射任务产生的中间键都有显著的重复，用户指定的Reduce函数是交换的和联想的。 这方面的一个很好的例子是2.1节中的单词计数示例。 由于单词频率倾向于遵循Zipf分布，因此每个地图任务将产生数百或数千个表单<记录，1>。所有这些计数将通过网络发送到一个减少任务，然后由减少函数添加到一起，生成一个数字。 我们允许用户指定一个可选的Combiner函数，在通过网络发送之前对这些数据进行部分合并。

在执行映射任务的每台机器上执行Combiner函数。 通常使用相同的代码来实现组合器和减少函数。 减少函数和组合函数的唯一区别是Map Reduce库如何处理函数的输出。 减少函数的输出写入最终输出文件。 组合器函数的输出被写入将发送到减少任务的中间文件。

部分组合大大加快了某些类Map Reduce操作。附录A包含一个使用组合器的示例。

4.4 输入和输出类型

Map Reduce库支持以多种不同格式读取输入数据。例如，“文本”

模式输入将每一行视为键/值对：键是文件中的偏移量，值是行的内容。 另一种常见的支持格式存储按键排序的键/值对序列。 每个输入类型实现都知道如何将自己分割成有意义的范围，作为单独的映射任务(例如。 文本模式的范围分裂确保范围分裂只发生在行边界)。 用户可以通过提供简单的读取器接口的实现来增加对新输入类型的支持，尽管大多数用户只是使用少量预定义的输入类型之一。

读取器不一定需要提供从文件读取的数据。 例如，很容易定义从数据库或内存中映射的数据结构读取记录的读取器。

以类似的方式，我们支持一组输出类型，用于以不同格式生成数据，用户代码很容易添加对新输出类型的支持。

4.5 副作用

在某些情况下，Map Reduce的用户发现，将辅助文件作为来自其映射和/或Reduce操作符的附加输出生成是方便的。 我们依靠应用程序编写器来使这种副作用具有原子性和幂等性。 通常，应用程序会写入一个临时文件，并在该文件完全生成后自动重命名。

我们不支持由单个任务产生的多个输出文件的原子两阶段提交。 因此，产生具有跨文件一致性要求的多个输出文件的任务应该是确定性的。 这一限制在实践中从未成为一个问题。

4.6 跳过坏记录

有时，用户代码中存在错误，导致Map或Reduce函数在某些记录上确实崩溃。 这些错误阻止了Map Reduce操作的完成。 通常的操作方法是修复bug，但有时这是不可行的；也许bug在源代码不可用的第三方库中。 此外，有时忽略一些记录是可以接受的，例如，在对大型数据集进行统计分析时。 我们提供了一种可选的执行模式，其中Map Reduce库检测哪些记录会导致确定性崩溃，并跳过这些记录，以便取得前进。

每个工作进程安装一个信号处理程序，它捕获分段违规和总线错误。 在调用用户Map或Reduce操作之前，Map Reduce库将参数的序列号存储在全局变量中。 如果用户代码产生信号，信号处理程序发送一个“最后一个GASP”UDP数据包，其中包含序列号到Map Reduce主程序。 当主机在特定记录上看到多个故障时，它表示当发出相应Map或Reduce任务的下一次重新执行时，应该跳过该记录。

4.7 本地执行

调试Map或Reduce函数中的问题可能很棘手，因为实际计算发生在分布式系统中，通常是在几千台机器上，工作分配决策是由主程序动态进行的。 为了帮助调试、分析和小规模测试，我们开发了Map Reduce库的替代实现，该实现顺序执行本地机器上Map Reduce操作的所有工作。 控件提供给用户，以便计算可以仅限于特定的映射任务。 用户用一个特殊的标志调用他们的程序，然后可以很容易地使用他们认为有用的任何调试或测试工具(例如。 全球开发银行)。

4.8 状况信息

主机运行内部HTTP服务器，并导出一组状态页，供人类消费。 状态页显示计算的进度，如完成了多少任务，进行中有多少任务，输入字节，中间数据字节，输出字节，处理速率等。 页面还包含指向每个任务生成的标准错误和标准输出文件的链接。 用户可以使用这些数据来预测计算需要多长时间，以及是否应该在计算中添加更多的资源。 这些页面还可以用来计算比预期慢得多的时间。

此外，顶层状态页显示哪些工作人员失败了，哪些映射和减少了失败时正在处理的任务。 当试图诊断用户代码中的错误时，此信息是有用的。

4.9 县

Map Reduce库提供了一个计数器工具来计数各种事件的发生。 例如，用户代码可能希望统计处理的单词总数或索引的德国文档的数量等。

要使用此工具，用户代码创建一个命名计数器对象，然后在Map和/或Reduce函数中适当地增加计数器。 例如：

```
计数器*大写;
大写=获取计数器（“大写”）;

映射（字符串名称，字符串内容）：对于内容中的
  每个单词w:
    如果(被资本化(W)：大写->增量
      ();
    中间(w, “1”);
```

来自单个工人机器的计数器值
周期性地传播到主服务器(基于ping响应)。 主程序从成功的映射中聚合计数器值并减少任务，并在Map Reduce操作完成后返回给用户代码。 当前计数器值也显示在主状态页上，这样人类就可以观看实时计算的进度。 在聚合计数器值时，主消除重复执行同一映射或减少任务的影响，以避免重复计数。 (重复执行可能产生于我们使用备份任务和由于故障而重新执行任务。)

一些计数器值由Map Reduce库自动维护，例如处理的输入键/值对的数量和产生的输出键/值对的数量。

用户已经发现计数器设施对于检查Map Reduce操作的行为是有用的。 例如，在一些Map Reduce操作中，用户代码可能希望确保产生的输出对的数量完全等于处理的输入对的数量，或者处理的德国文档的分数在处理的文档总数的某个可容忍的分数之内。

5 业绩

在本节中，我们在运行在大型机器集群上的两个计算上测量Map Reduce的性能。 一个计算搜索大约1兆字节的数据，寻找一个特定的模式。 另一个计算对大约1兆字节的数据进行排序。

这两个程序代表了由Map Reduce用户编写的真实程序的一个大子集--一类程序将数据从一个表示洗牌到另一个表示，另一类程序从一个大型数据集提取少量有趣的数据。

5.1 集群配置

所有的程序都是在一个由大约1800台机器组成的集群上执行的。 每台机器都有两个2GHz英特尔Xeon处理器，启用超线程，4GB内存，两个160GB IDE

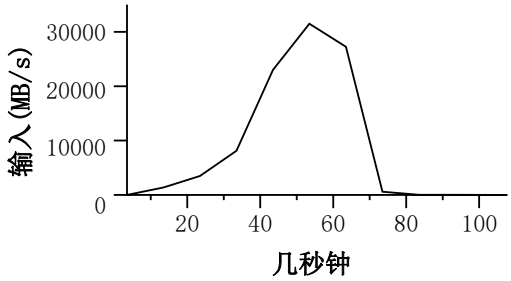


图2：数据传输速率随时间的变化

磁盘和千兆以太网链路。 这些机器被安排在一个两级树状交换网络中，其根部有大约100-200Gbps的总带宽。 所有的机器都在同一个主机设施中，因此任何一对机器之间的往返时间都小于毫秒。

在4GB的内存中，大约1-1.5GB被运行在集群上的其他任务保留。 这些程序是在一个周末的下午执行的，那时CPU、磁盘和网络大多处于空闲状态。

5.2 盖普

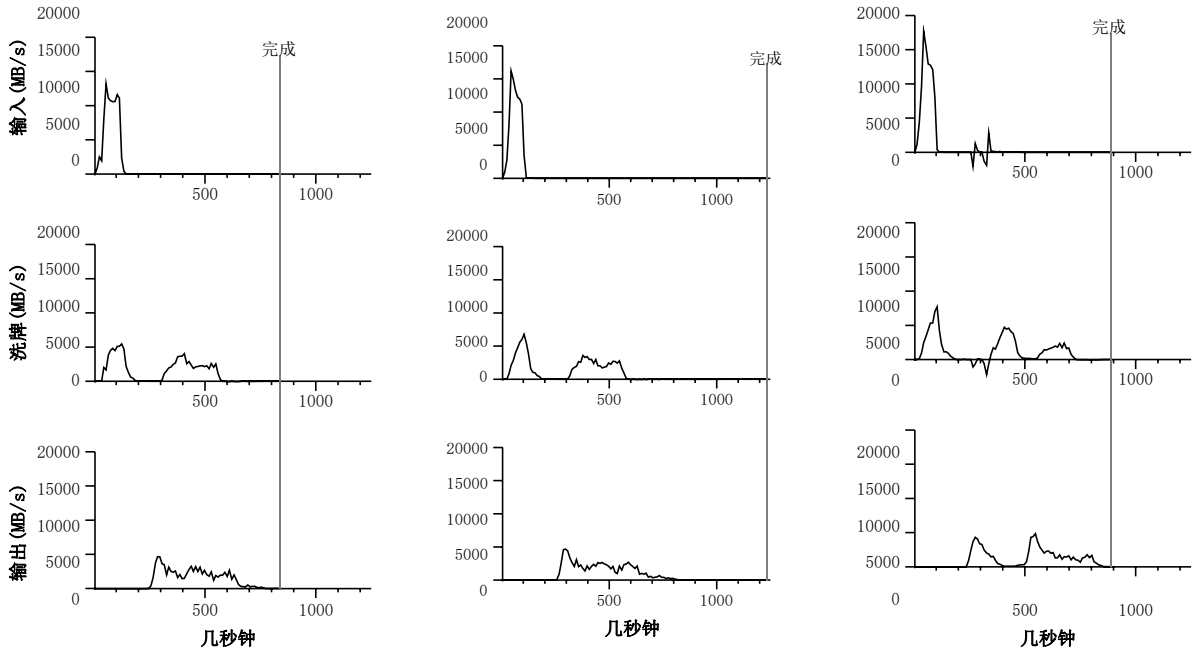
grep程序扫描10^{10}个100字节记录，搜索一个相对罕见的三字符模式（该模式发生在92，337条记录中）。 输入被分成大约64MB的部分(M=15000)，整个输出被放置在一个文件中(R=1)。

图2显示了随着时间的推移计算的进度。 Y轴显示输入数据扫描的速率。 随着更多的机器被分配到这个Map Reduce计算中，这个速率逐渐上升，当分配了1764名工人时，峰值在30GB/s以上。 当地图任务完成后，速率开始下降，并在计算中击中零约80秒。 整个计算从开始到结束大约需要150秒。 这包括大约一分钟的启动开销。 开销是由于程序传播到所有工作机器，并且延迟与GFS交互以打开1000个输入文件集并获取局部优化所需的信息。

5.3 算是吧

排序程序排序10^{10} 100字节记录（约1兆字节数据）。 此程序是根据Tera排序基准[10]建模的。

排序程序由少于50行用户代码组成。 三线地图函数从文本行中提取10字节排序键并发出键和



(a) 正常执行　　(b) 没有备份任务　　(c) 200个任务被杀

图3：不同执行排序程序的数据传输速率随时间的变化

原始文本行作为中间键/值对。 我们使用内置的标识函数作为减少运算符。 此函数传递中间键/值对，与输出键/值对不变。 最终排序的输出被写入一组双向复制的GFS文件(即2TB被写入程序的输出)。

与以前一样，输入数据被分成64MB(M=15000)。我们将排序后的输出划分为4000个文件(R=4000)。分区函数使用密钥的初始字节将其分隔为R件之一。

我们为这个基准的分区函数内置了密钥分布的知识。 在一般排序程序中，我们将添加一个预通Map Reduce操作，该操作将收集密钥的样本，并使用采样密钥的分布来计算最终排序通过的分裂点。

图3(a)显示了排序程序正常执行的进度。 左上图显示了读取输入的速率。 由于所有地图任务都在200秒之前完成，所以速率峰值在13GB/s左右，并且很快消失。 注意，输入速率小于grep。 这是因为排序映射任务花费了大约一半的时间，I/O带宽将中间输出写入本地磁盘。 相应的grep中间输出的尺寸可以忽略不计。

中左图显示从映射任务到减少任务的数据通过网络发送的速率。 这一洗牌一旦第一个地图任务完成就开始。 图中的第一个驼峰是为了第一批大约1700个减少任务(整个Map Reduce被分配了大约1700台机器，每台机器一次最多执行一个减少任务)。 大约300秒进入计算，其中一些第一批减少任务完成，我们开始为剩余的减少任务洗牌数据。 所有的洗牌都是在600秒左右完成的。 左下角的图表显示了排序数据被还原任务写入最终输出文件的速率。 在第一个洗牌期结束和写入期开始之间存在延迟，因为机器正在忙着对中间数据进行排序。 该写入以2-4GB/s的速率持续一段时间。 所有的写完成大约850秒的计算。 包括启动开销，整个计算需要891秒。 这类似于Tera排序基准[18]当前1057秒的最佳报告结果。 需要注意的是：由于我们的局部优化，输入速率高于洗牌速率和输出速率-大多数数据是从本地磁盘读取的，并且绕过我们相对带宽受限的网络。 洗牌率高于输出率，因为输出阶段写入了排序数据的两个副本（出于可靠性和可用性原因，我们对输出进行了两个副本）。 我们编写两个副本，因为这是底层文件系统提供的可靠性和可用性机制。 如果底层文件系统使用擦除编码[14]而不是擦除编码，则编写数据的网络带宽需求将减少复制。

5.4 备份任务的影响

在图3(b)中，我们展示了禁用备份任务的排序程序的执行。 执行流程类似于图3(a)中所示，只是有一个很长的尾部，几乎没有任何写入活动发生。 在960秒后，除5个减少任务外，所有任务都完成。 然而，最后几个散兵直到300秒后才结束。 整个计算需要1283秒，在经过的时间内增加了44。

5.5 机器故障

在图3(C)中，我们显示了排序程序的执行，在该程序中，我们故意在计算过程中几分钟内杀死了1746个工作人员中的200个。 底层集群调度程序立即在这些机器上重新启动新的工作进程（因为只有进程被杀死，机器仍然正常工作）。

工人死亡显示为负输入率，因为一些以前完成的地图工作消失了（因为相应的地图工人被杀），需要重做。 此映射工作的重新执行相对较快。 整个计算在933秒内完成，包括启动开销（仅比正常执行时间增加5）。

6 经验

我们在2003年2月编写了Map Reduce库的第一个版本，并在2003年8月对其进行了重大改进，包括局部优化、跨工机任务执行的动态负载平衡等。 自那时以来，我们一直令人愉快地感到惊讶的是，Map Reduce库对于我们所处理的各种问题多广泛的适用性。 它已经在谷歌内部的广泛领域中使用，包括：

- 大规模机器学习问题，
- 谷歌新闻和Froogle产品的集群问题，
- 提取用于生成流行查询报告的数据(例如。Google Zeitgeist)，
- 为新的实验和产品提取网页的属性(例如。 从大量的网页中提取地理位置，以便进行本地化搜索)，以及
- 大规模图形计算。

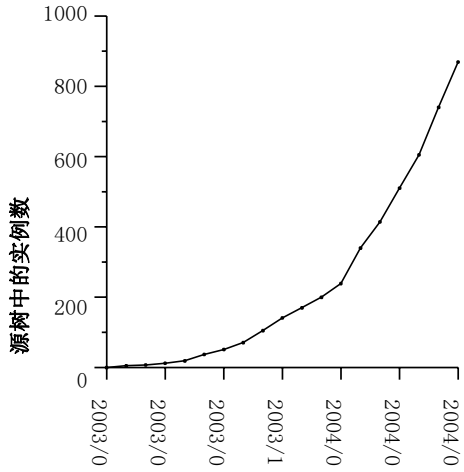


图4：随着时间的推移，映射减少实例

就业人数	29,423
平均工作完成时间使用机器天数	634秒
	79186天
输入数据读取	3,288TB
中间数据产生输出数据写入	758TB
	193TB
平均每项工作的工人机器	157
平均每项工作的工人死亡	1.2
每个作业的平均地图任务	3,351
平均减少每个工作的任务	55
独特的地图实现	395
独特的减少实现独特的地图/减少组合	269
	426

表1：2004年8月减少就业的地图

图4显示了随着时间的推移，在我们的主源代码管理系统中检查的单独Map Reduce程序的数量显著增加，从2003年初的0个增加到2004年9月底的近900个单独实例。 Map Reduce非常成功，因为它可以在半个小时内编写一个简单的程序并在千台机器上高效运行，大大加快了开发和原型开发周期。 此外，它允许没有分布式和/或并行系统经验的程序员轻松地利用大量资源。

在每个作业的末尾，Map Reduce库记录有关作业使用的计算资源的统计信息。 在表1中，我们展示了2004年8月在谷歌运行的Map Reduce作业子集的一些统计数据。

6.1 大规模索引

到目前为止，Map Reduce的一个最重要的用途是完全重写生产索引-

生成用于谷歌网络搜索服务的数据结构的ING系统。索引系统以我们的爬行系统检索到的一组大型文档作为输入，存储为一组GFS文件。 这些文档的原始内容是20TB以上的数据。 索引过程作为五到十个Map Reduce操作的序列运行。 使用Map Reduce（而不是索引系统的前一个版本中的临时分布式传递）提供了几个好处：

- 索引代码更简单、更小、更容易理解，因为处理容错、分发和并行化的代码隐藏在Map Reduce库中。 例如，当使用Map Reduce表示时，计算的一个阶段的大小从大约3800行C++代码下降到大约700行。
- Map Reduce库的性能足够好，我们可以将概念上无关的计算分开，而不是将它们混合在一起，以避免额外的数据传递。 这使得索引过程很容易改变。 例如，在我们的旧索引系统中进行的一次更改只花了几个月的时间，就在新系统中实现了。
- 索引过程变得更容易操作，因为大多数由机器故障、慢机器和网络故障引起的问题都由Map Reduce库自动处理，而无需操作员干预。 此外，通过在索引集群中添加新的机器，很容易提高索引过程的性能。

7 相关工作

许多系统提供了受限的编程模型，并利用这些限制自动并行计算。 例如，使用并行前缀计算[6,9,13]，可以在N个处理器上的日志N时间中对N个元素数组的所有前缀计算关联函数。 地图减少可以被认为是一种简化和蒸馏这些模型的一些基础上，我们的经验，大的实际计算。 更重要的是，我们提供了一个容错实现，扩展到数千个处理器。 相反，大多数并行处理系统只在较小的规模上实现，并将处理机器故障的细节留给程序员。

批量同步编程[17]和一些MPI原语[11]提供更高层次的抽象使程序员更容易编写并行程序。 这些系统和Map Reduce之间的一个关键区别是Map Reduce利用一个受限的编程模型自动并行化用户程序并提供透明的容错性。

我们的局部性优化从主动磁盘[12,15]等技术中汲取灵感，在这些技术中，计算被推到接近局部磁盘的处理元素中，以减少跨I/O子系统或网络发送的数据量。 我们运行在商品处理器上，其中少量磁盘直接连接，而不是直接运行在磁盘控制器处理器上，但一般的方法是相似的。

我们的备份任务机制类似于夏洛特系统[3]中使用的渴望调度机制。 简单渴望调度的缺点之一是，如果给定的任务导致重复故障，整个计算就无法完成。 我们用跳过坏记录的机制来修复这个问题的一些实例。

Map Reduce实现依赖于一个内部集群管理系统，该系统负责在大量共享机器上分配和运行用户任务。 集群管理系统虽然不是本文的重点，但在精神上与Condor[16]等其他系统相似。

作为Map Reduce库的一部分的排序设施在运行中类似于NOW-Sort[1]。 源机（地图工）对要排序的数据进行分区，并将其发送给R减料工之一。 每个reduce工作人员对其数据进行本地排序（如果可能的话在内存中）。 当然，NOW-Sort没有使我们的库广泛适用的用户可定义的Map和Reduce函数。

River[2]提供了一个编程模型，其中进程通过分布式队列发送数据相互通信。 与Map Reduce一样，River系统试图提供良好的平均案例性能，即使在异构硬件或系统扰动引入的非均匀性的情况下也是如此。 河通过仔细调度磁盘和网络传输来实现这一点，以实现平衡的完成时间。 Map Reduce有不同的方法。 通过限制编程模型，Map Reduce框架能够将问题划分为大量细粒度任务。 这些任务动态地安排在可用的工人上，以便更快的工人处理更多的任务。 受限编程模型还允许我们在工作接近结束时对任务进行冗余执行，这大大减少了在不一致情况下（如缓慢或卡住的工人）的完成时间）。

BAD-FS[5]有一个与Map Reduce非常不同的编程模型，与Map Reduce不同，它的目标是

跨广域网执行作业。 然而，有两个基本的相似之处。 （1）两个系统都使用冗余执行从故障造成的数据损失中恢复。 （2）两者都使用位置感知调度来减少通过拥挤的网络链路发送的数据量。

TACC[7]是一个简化高可用网络服务结构的系统。与Map Reduce一样，它依赖于重新执行作为实现容错的机制。

8 结论

Map Reduce编程模型已在谷歌成功地用于许多不同的目的。 我们把这种成功归因于几个原因。 首先，该模型易于使用，即使对于没有并行和分布式系统经验的程序员也是如此，因为它隐藏了并行化、容错、局部性优化和负载平衡的细节。 其次，在Map Reduce计算中，许多问题都很容易解释。 例如，Map Reduce用于为Google的生产Web搜索服务、排序、数据挖掘、机器学习和许多其他系统生成数据。 第三，我们开发了一个Map Reduce的实现，将其扩展到由数千台机器组成的大型机器集群。 该实现有效地利用了这些机器资源，因此适合在谷歌遇到的许多大型计算问题上使用。

我们从这项工作中学到了几件事。 首先，限制编程模型使计算易于并行和分发，并使这种计算具有容错性。 第二，网络带宽是一种稀缺资源。 因此，我们系统中的一些优化旨在减少跨网络发送的数据量：局部性优化允许我们从本地磁盘读取数据，并将中间数据的单个副本写入本地磁盘节省了网络带宽。 第三，冗余执行可以减少慢机的影响，处理机器故障和数据丢失。

致谢

Josh Levenberg在修改和扩展用户级的Map Reduce API方面发挥了重要作用，根据他使用Map Reduce和其他人对增强的建议的经验，他提供了一些新的功能。 Map Reduce从Google文件系统[8]读取其输入并将其输出写入Google文件系统。 我们要感谢莫希特·阿隆、霍华德·戈比奥夫、马库斯·古奇克，大卫·克莱默、梁顺泰和乔希·雷德斯通在开发GFS方面的工作。 我们还要感谢PercyLang和OlcanSercinoglu在开发Map Reduce使用的集群管理系统方面所做的工作。 迈克·布伦斯、威尔逊·谢赫、乔希·莱文伯格、莎伦·佩尔、罗伯·派克和DebbyWallach对本文的早期草稿提供了有益的评论。 匿名的OSDI评审人员和我们的牧羊人EricBrewer提供了许多有用的建议，说明了论文可以改进的领域。 最后，我们感谢Google工程组织内所有Map Reduce的用户提供有用的反馈、建议和错误报告。

参考资料

[1] Andrea C.Arpaci-Dusseau，Remzi H.Arpaci-Dusseau，David E。 卡尔勒，约瑟夫M。 海勒斯坦和大卫·A。 帕特森。 工作站网络的高性能排序。 1997年在亚利桑那州图森举行的ACM SIGMOD国际数据管理会议记录，1997年5月。

[2] Remzi H.Arpaci-Dusseau、Eric Anderson、Noah Treuhaft、David E.Culler、Joseph M.Hellerstein、David Patterson和Kathy Yelick。 带河流的I/O集群：使快速案例变得常见。 第六次并行和分布式系统输入/输出讲习班记录(IOPADS‘99)，第10-22页，佐治亚州亚特兰大，1999年5月。

[3] Arash Baratloo、Mehmet Karaul、Zvi Kedem和Peter Wyckoff。 夏洛特：网上的电脑。 *1996年第9届并行和分布式计算系统国际会议记录。*

[4] Luiz A。 巴罗佐，杰弗里·迪恩和乌尔斯·豪¨伊兹尔。 搜索星球：谷歌集群架构。 IEEE Micro，23（2）：22-28，2003年4月。

[5] John Bent、Douglas Thain、Andrea C.Arpaci-Dusseau、Remzi H.Arpaci-Dusseau和Miron Livny。 批处理感知的分布式文件系统中的显式控制。 在第一届USENIX网络系统设计与实现国家空间数据基础设施研讨会论文集，2004年3月。

[6] 盖伊·E·布莱洛克。 扫描作为原始并行操作。 *IEEE计算机交易，C-38（11），1989年11月。*

[7] Armando Fox，Steven D.Gribble，Yatin Chawathe，Eric A。 布鲁尔和保罗·高瑟。 基于集群的可伸缩网络服务。 第16届ACM操作系统原则研讨会论文集，第78-91页，法国圣马洛，1997年。

[8] 桑杰·吉马瓦特、霍华德·戈比奥夫和梁顺塔克。 谷歌文件系统。 第19届操作系统原则研讨会，第29至43页，乔治湖，纽约，2003年。

[9] S.戈拉奇。 扫描和其他列表同态的系统有效并行化。 在L.Bouge，P。Fraigniaud，A.Mignotte和Y。罗伯特，编辑，Euro-Par‘96。 并行处理，计算机科学讲座笔记1124，第401-408页。 斯普林格-维拉格，1996年。

[10] Jim 格雷。排序基准首页。 http://research.microsoft.com/barc/SortBenchmark/。

[11] 威廉·格罗普、尤因·卢斯克和安东尼·斯凯勒姆。 *使用MPI：带有消息传递接口的可移植并行编程。 麻省理工学院出版社，剑桥，MA，1999年。*

[12] l. 休斯顿，R。 苏库纳，R。 Wickremesinghe，M。 Satyanarayanan，G。 r。 甘格，E。 里德尔和A。 艾拉马基。 钻石：用于交互式搜索中早期丢弃的存储架构。 2004年USENIX文件和存储技术快速会议记录，2004年4月。

[13] 理查德·拉德纳和迈克尔·J·费舍尔。 并行前缀计算。 *ACM杂志，27（4）：831-838，1980年。*

[14] 迈克尔·奥·拉宾。 信息的高效分散，用于安全，负载均衡和容错。 ACM杂志，36（2）：335-348，1989年。

[15] 埃里克·里德尔、克里斯托斯·法鲁索斯、加思·A·吉布森和大卫·纳格尔。 用于大规模数据处理的活动磁盘。 IEEE计算机，第68-74页，2001年6月。

[16] 道格拉斯·塞恩，托德·坦南鲍姆和米隆·利夫尼。 分布式计算在实践中：秃鹰的经验。 *并发和计算：实践和经验，2004年。*

[17] l.G。勇敢。 并行计算的桥接模型。 *ACM的来文，33（8）：103-111，1997年。*

[18] 吉姆·怀利。 斯皮索：如何快速排序一个TB。 http://almel.almaden.ibm.com/cs/spsort.pdf.

一个字频率

本节包含一个程序，它计算命令行上指定的一组输入文件中每个唯一单词的出现次数。

```
#包括“mapreduce/mapreduce.h”

//用户的地图功能
类字计数器：公共映射器{公共：
  虚拟虚空映射(ConstMapInput&Input){const字符串
    &text=input.value(); constintn=text.size();
    对于(inti=0; i<n; ){
      //跳过前导空白，而(I<n)&isspace(文本[i])
        我++;

      //找到词尾，开始=我；
      而(i<n)&! isspace(文本[i])我++;
        如果（从第一<开始）
          Emit(text.substr(start，i-start)，“1”)
    }
  }
};
REGISTER_MAPPER（文字计数器）；

//用户的减少函数类加法器：公共减速
器{
            虚拟无效减少（减少输入*输入）{
                  //迭代所有条目
    //相同的键，并将值int64值添加=0；
    而（！输入->()完成）{
      值+=String to Int（输入->值()）；输入->下一
      个值；
    }

    //输入->键的Emit和()Emit(Intto String（值））；
  }
}；REGISTER_REDUCER（加法器）；

int main(int argc，char**argv){Parse命令行Flags(argc，
  argv)；

  地图减少规格规范；

  //将输入文件列表存储到(inti=1；i<argc；i++){的
  “规范”中){
    映射减少输入*输入=规范add_input()；输入-
    >set_format（“文本”）；
    输入->set_filepattern(argv[i])
    输入->set_mapper_class（“文字计数器”）；
  }

  //指定输出文件：
  //     /gfs/test/freq-00000-of-00100
  //     /gfs/test/freq-00001-of-00100
  //     ...
  Map Reduce Output*out=spec.Output()；out-
  >set_filebase(“/gfs/test/freq”)；out-
  >set_filebaseset_num_tasks（100）；
  >set_format（“案文”）；
  >set_reducer_class（“加法器”）；

  //可选：在地图内做部分和
  //任务，以节省网络带宽>set_combiner_class
  （“加法器”）；

  //调谐参数：最多使用2000
  每个任务规范//机器和100MB内存。set_machines（2000）；规
  范set_map_megabytes（100）；规范set_reduce_megabytes
  （100）；

  //现在运行它Map Reduce Result；
  如果(！ 地图减少（规范，和结果)中止()；

  //完成：“结果”结构包含信息
  //柜台，时间，数量
  用的//机器等。

  返回0；
}
```